

Scalable Continual Top- k Keyword Search in Relational Databases

Yanwei Xu¹

Department of Computer Science and Technology, Tongji University, Shanghai, China

Abstract. *Keyword search in relational databases* has been widely studied in recent years because it does not require users neither to master a certain structured query language nor to know the complex underlying database schemas. Most of existing methods focus on answering *snapshot keyword queries* in static databases. In practice, however, databases are updated frequently, and users may have long-term interests on specific topics. To deal with such a situation, it is necessary to build effective and efficient facility in a database system to support *continual keyword queries*.

In this paper, we propose an efficient method for answering continual top- k keyword queries over relational databases. The proposed method is built on an existing scheme of keyword search on relational data streams, but incorporates the ranking mechanisms into the query processing methods and makes two improvements to support efficient top- k keyword search in relational databases. Compared to the existing methods, our method is more efficient both in computing the top- k results in a static database and in maintaining the top- k results when the database continually being updated. Experimental results validate the effectiveness and efficiency of the proposed method.

Key words: Relational databases, keyword search, continual queries, results maintenance.

1 Introduction

With the proliferation of text data available in relational databases, simple ways to exploring such information effectively are of increasing importance. *Keyword search in relational databases*, with which a user specifies his/her information need by a set of keywords, is a popular information retrieval method because the user needs to know neither a complex query language nor the underlying database schemas. It has attracted substantial research effort in recent years, and a number of methods have been developed [1,2,3,4,5,6,7,8,9,10].

Example 1. Consider a sample publication database shown in Fig. 1. Fig. 1 (a) shows the three relations *Papers*, *Authors*, and *Writes*. In the following, we use the initial of each relation name (P , A , and W) as its shorthand. There are two foreign key references: $W \rightarrow A$ and $W \rightarrow P$. Fig. 1 (b) illustrates the tuple connections based on the foreign key

references. For the keyword query “James P2P” consisting of two keywords “James” and “P2P”, there are six tuples in the database that contain at least one of the two keywords (underlined in Fig. 1 (a)). They can be regraded as the results of the query. However, they can be joined with other tuples according to the foreign key references to form more meaningful results, several of which are shown in Fig. 1 (c). The arrows represent the foreign key references between the corresponding pairs of tuples. Finding such results which are formed by the tuples containing the keywords is the task of keyword search in relational databases. As described later, results are often ranked by relevance scores evaluated by a certain ranking strategy. \square

Papers

<i>pid</i>	<i>title</i>
p_1	“Leveraging Identity-Based Cryptography for Node ID Assignment in Structured P2P Systems.”
p_2	“P2P or Not P2P?: In P2P 2003”
p_3	“A System for Predicting Subcellular Localization.”
p_4	“Logical Queries over Views: Decidability.”
p_5	“A conservative strategy to protect P2P file sharing systems from pollution attacks.”
...	...

Authors

<i>aid</i>	<i>name</i>
a_1	“James Chen”
a_2	“Saikat Guha”
a_3	“James Basingthwaite”
a_4	“Sabu T.”
a_5	“James S. W. Walkerdines”
...	...

Writes

<i>wid</i>	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	...
<i>aid</i>	a_1	a_2	a_3	a_1	a_5	a_3	a_2	a_2	...
<i>pid</i>	p_2	p_1	p_3	p_4	p_5	p_4	p_2	p_5	...

(a) Database (Matched keywords are underlined)

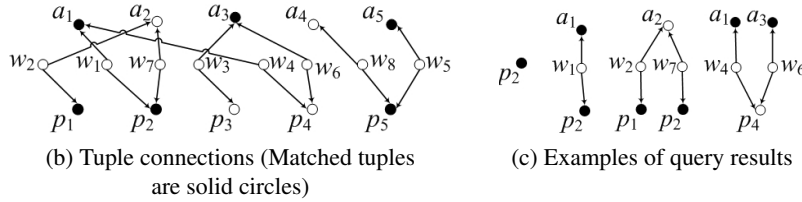


Fig. 1. A sample database with a keyword query “James P2P”.

Most of the existing keyword search methods assume that the databases are static and focus on answering *snapshot* keyword queries. In practice, however, a database is often updated frequently, and the result of a snapshot query becomes invalid once the related data in the database is updated. For the database in Fig. 1, if publication data comes continually, new publication records are inserted to the three tables. Such new records may be more relevant to “James” and “P2P”. Hence, after getting the initial top- k results, the user may demand the top- k results to reflect the latest database updates.

Such demands are common in real applications. Suppose a user want to do a top- k keyword search in a Micro-blogging database, which is being updated continually: not only the weblogs and comments are continually being inserted or deleted by bloggers, but also the follow relationship between bloggers are being updated continually. Thus, a continual evaluation facility for keyword queries is essential in such databases.

For continual keyword query evaluation, when the database is updated, two situations must be considered:

1. Database updates may change the existing top- k results: some top- k results may be replaced by new ones that are related to the new tuples, and some top- k results may be invalid due to deletions.
2. Database updates may change the relevance scores of existing results because the underlying statistics (e.g., word frequencies) are changed.

In this paper, we describe a system which can efficiently report the top- k results of every monitoring query while the database is being updated continually. The outline of the system is as follows:

- When a continual query is issued, it is evaluated in a pipelined way to find the set of results whose upper bounds of relevance scores are higher than a threshold θ by calculating the upper bound of the future relevance score for every query result.
- When the database is updated, we first update the relevance scores of the computed results, then find the new results whose upper bounds of relevance scores are larger than θ and delete the results containing the deleted tuples.
- The pipelined evaluation of the keyword query is resumed if the number of computed results whose relevance scores are larger than θ falls below k , or is reversed if the above number is much larger than k .
- At any time, the k computed results whose relevance scores are the largest and are larger than θ are reported as the top- k results.

In Section 2, some basic concepts are introduced and the problem is defined. Section 3 discusses related work. Section 4 presents the details of the proposed method. Section 5 gives the experimental results. Conclusion is drawn in Section 6.

2 Preliminaries

In this section, we introduce some important concepts for top- k keyword querying evaluation in relational databases.

2.1 Relational Database Model

We consider a relational database schema as a directed graph $G_S(V, E)$, called a schema graph, where V represents the set of relation schemas $\{R_1, R_2, \dots\}$ and E represents the foreign key references between pairs of relation schemas. Given two relation schemas,

R_i and R_j , there exists an edge in the schema graph, from R_j to R_i , denoted $R_i \leftarrow R_j$, if the primary key of R_i is referenced by the foreign key defined on R_j . For example, the schema graph of the publication database in Fig. 1 is $Papers \leftarrow Write \rightarrow Authors$. A relation on relation schema R_i is an instance of R_i (a set of tuples) conforming to the schema, denoted $r(R_i)$. A tuple can be inserted into a relation. Below, we use R_i to denote $r(R_i)$ if the context is obvious.

2.2 Joint-Tuple-Trees (JTTs)

The results of keyword queries in relational databases are a set of connected trees of tuples, each of which is called a *joint-tuple-tree* (JTT for short). A JTT represents how the *matched tuples*, which contain the specified keywords in their text attributes, are interconnected through foreign key references. Two adjacent tuples of a JTT, $t_i \in r(R_i)$ and $t_j \in r(R_j)$, are interconnected if they can be joined based on a foreign key reference defined on relational schema R_i and R_j in G_S (either $R_i \rightarrow R_j$ or $R_i \leftarrow R_j$). The foreign key references between tuples in a JTT can be denoted using arrows or notation \bowtie . For example, the second JTT in Fig. 1(c) can be denoted as $a_1 \leftarrow w_1 \rightarrow p_2$ or $a_1 \bowtie w_1 \bowtie p_2$. To be a valid result of a keyword query Q , each leaf of a JTT is required to contain at least one keyword of Q . In Fig. 1(c), tuples p_1, p_2, a_1 and a_3 are matched tuples to the keyword query as they contain the keywords. Hence, the four JTTs are valid results to the query. In contrast, $p_1 \leftarrow w_2 \rightarrow a_2$ is not a valid result because tuple a_2 does not contain any required keywords. The number of tuples in a JTT T is called the *size* of T , denoted by $size(T)$.

2.3 Candidate Networks (CNs)

Given a keyword query Q , the *query tuple set* R_i^Q of relation R_i is defined as $R_i^Q = \{t \in r(R_i) \mid t \text{ contains some keywords of } Q\}$. For example, the two query tuple sets in Example 1 are $P^Q = \{p_1, p_2, p_5\}$ and $A^Q = \{a_1, a_3, a_5\}$, respectively. The *free tuple set* R_i^F of a relation R_i with respect to Q is defined as the set of tuples that do not contain any keywords of Q . In Example 1, $P^F = \{p_3, p_4, \dots\}$, $A^F = \{a_2, a_4, \dots\}$. If a relation R_i does not contain text attributes (e.g., relation W in Fig. 1), R_i is used to denote R_i^F for any keyword query. We use R_i^{QorF} to denote a *tuple set*, which may be either R_i^Q or R_i^F .

Each JTT belongs to the result of a relational algebra expression, which is called a *candidate network* (CN) [4,9,11]. A CN is obtained by replacing each tuple in a JTT with the corresponding tuple set that it belongs to. Hence, a CN corresponds to a join expression on tuple sets that produces JTTs as results, where each join clause $R_i^{QorF} \bowtie R_j^{QorF}$ corresponds to an edge $\langle R_i, R_j \rangle$ in the schema graph G_S , where \bowtie represents a equi-join between relations. For example, the CNs that correspond to two JTTs p_2 and $p_2 \leftarrow w_1 \rightarrow a_1$ in Example 1 are P^Q and $P^Q \bowtie W \bowtie A^Q$, respectively. In the following, we also denote $P^Q \bowtie W \bowtie A^Q$ as $P^Q \leftarrow W \rightarrow A^Q$. As the leaf nodes of JTTs must be matched tuples, the leaf nodes of CNs must be query tuple sets. Due to the existence of $m : n$ relationships (for example, an article may be written by multiple authors), a CN may have multiple occurrences of the same tuple set. The *size* of CN C , denoted as

$size(C)$, is defined as the number of tuple sets that it contains. Obviously, the size of a CN is the same as that of the JTTs it produces. Fig. 2 shows the CNs corresponding to the four JTTs shown in Fig. 1 (c). A CN can be easily transformed into an equivalent SQL statement and executed by an RDBMS.¹

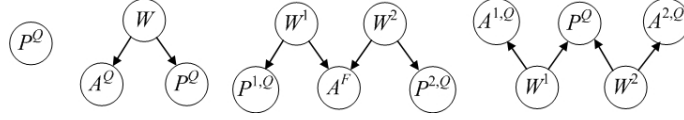


Fig. 2. Examples of Candidate Networks

When a continual keyword query $Q = \{w_1, w_2, \dots, w_l\}$ is specified, the non-empty query tuple set R_i^Q for each relation R_i in the target database are firstly computed using full-text indices. Then all the non-empty query tuple sets and the database schema are used to generate the set of valid CNs, whose basic idea is to expand each partial CN by adding a R_i^F or R_i^Q at each step (R_i is adjacent to one relation of the partial CN in G_S), beginning from the set of non-empty query tuple sets. The set of CNs shall be sound/complete and duplicate-free. There are always a constraint, CN_{\max} (the maximum size of CNs) to avoid generating complicated but less meaningful CNs. In the implementation, we adopt the state-of-the-art CN generation algorithm proposed in [12].

Example 2. In Example 1, there are two non-empty query tuple sets P^Q and A^Q . Using them and the database schema graph, if $CN_{\max} = 5$, the generated CNs are: $CN_1 = P^Q$, $CN_2 = A^Q$, $CN_3 = P^Q \leftarrow W \rightarrow A^Q$, $CN_4 = P^Q \leftarrow W \rightarrow A^Q \leftarrow W \rightarrow P^Q$, $CN_5 = P^Q \leftarrow W \rightarrow A^F \leftarrow W \rightarrow P^Q$, $CN_6 = A^Q \leftarrow W^1 \rightarrow P^Q \leftarrow W \rightarrow A^Q$ and $CN_7 = A^Q \leftarrow W \rightarrow P^F \leftarrow W \rightarrow A^Q$.

2.4 Scoring Method

The problem of *continual top- k keyword search* we study in this paper is to continually report top- k JTTs based on a certain scoring function that will be described below. We adopt the scoring method employed in [4], which is an ordinary ranking strategy in the information retrieval area. The following function $score(T, Q)$ is used to score JTT T for query Q , which is based on the TF-IDF weighting scheme:

$$score(T, Q) = \frac{\sum_{t \in T} tscore(t, Q)}{size(T)}, \quad (1)$$

where $t \in T$ is a tuple (a node) contained in T . $tscore(t, Q)$ is the *tuple score* of t with regard to Q defined as follows:

$$tscore(t, Q) = \sum_{w \in t \cap Q} \frac{1 + \ln(1 + \ln(tf_{t,w}))}{(1 - s) + s \cdot \frac{dl_t}{avdl}} \cdot \ln\left(\frac{N}{df_w + 1}\right) \quad (2)$$

¹ For example, we can transform CN $P^Q \leftarrow W \rightarrow A^Q$ as: SELECT * FROM W w, P p, A a WHERE w.pid = p.pid AND w.aid = a.aid AND p.pid in (p_1, p_2, p_5) and a.aid in (a_1, a_3, a_5).

where $tf_{t,w}$ is the *term frequency* of keyword W in tuple t , df_w is the number of tuples in relation $r(t)$ (the relation corresponds to tuple t) that contain W . df_w is interpreted as the *document frequency* of W . dl_t represents the size of tuple t , i.e., the number of letters in t , and is interpreted as the *document length* of t . N is the *total number* of tuples in $r(t)$, $avdl$ is the *average tuple size (average document length)* in $r(t)$, and s ($0 \leq s \leq 1$) is a constant which usually be set to 0.2.

Table 1 shows the tuple scores of the six matched tuples in Example 1. We suppose all the matched tuples are shown in Fig. 1, and the numbers of tuples of the two relations are 150 and 180, respectively. Therefore, the top-3 results are $T_1 = p_2$ (*score* = 7.04), $T_2 = a_1$ (*score* = 4.00) and $T_3 = p_2 \leftarrow w_1 \rightarrow a_1$ (*score* = 3.68).

Table 1. Statistics and tuple scores of tuples of P^Q and A^Q

Tuple Set	P^Q			A^Q		
	N	df_{p2p}	$avdl$	N	df_{James}	$avdl$
Statistics	150	3	57.8	170	3	14.6
Tuple	p_1	p_2	p_5	a_1	a_3	a_5
dl	88	28	83	10	22	23
tf	1	3	1	1	1	1
$tscore$	3.28	7.04	3.33	4.00	3.40	3.36

The score function in Eq. (1) has the property of *tuple monotonicity*, defined as follows. For any two JTTs $T = t_1 \bowtie t_2 \bowtie \dots \bowtie t_l$ and $T' = t'_1 \bowtie t'_2 \bowtie \dots \bowtie t'_l$ generated from the same CN C , if for any $1 \leq i \leq l$, $tscore(t_i, Q) \leq tscore(t'_i, Q)$, then we have $score(T, Q) \leq score(T', Q)$. As shown in the following discussion, this property is relied by the existing top- k query evaluation algorithms.

3 Related Work

3.1 Keyword Search in Relational Databases

Given l -keyword query $Q = \{w_1, w_2, \dots, w_l\}$, the task of keyword search in a relational database is to find structural information constructed from tuples in the database [13]. There are two approaches. The *schema-based approaches* [1,2,4,7,9,14,15] in this area utilize the database schema to generate SQL queries which are evaluated to find the structures for a keyword query. They process a keyword query in two steps. They first utilize the database schema to generate a set of relation join templates (i.e., the CNs), which can be interpreted as select-project-join views. Then, these join templates are evaluated by sending the corresponding SQL statements to the DBMS for finding the query results. [2] proved how to generate a complete set of CNs when the CN_{\max} has a user-given value and discussed several query processing strategies when considers the common sub-expressions among the CNs. [1,2,14,15] all focused on finding all JTTs, whose sizes are $\leq CN_{\max}$, which contain all l keywords, and there is no ranking

involved. In [4] and [9], several algorithms are proposed to get top- k JTTs. We will introduce them in detail in Section 3.2.

The *graph-based methods* [3,8,5,6,10,16] model and materialize the entire database as a directed graph where the nodes are relational tuples and the directed edges are foreign key references between tuples. Fig. 1(b) shows such a database graph of the example database. Then for each keyword query, they find a set of structures (either Steiner trees [3], distinct rooted trees [5], r -radius Steiner graphs [10], or multi-center subgraphs [16]) from the database graph, which contain all the query keywords and are connected by the paths in database graph. Such results are found by graph traversals that start from the nodes that contain the keywords. For the details, please refer the survey papers [13,17]. The materialized data graph should be updated for any database changes; hence this model is not appropriate to the databases that change frequently [17]. Therefore, this paper adopts the schema-based framework and can be regarded as an extension for dealing with continual keyword search.

3.2 Top- k Keyword Search in Relational Databases

DISCOVER2 [4] proposed the *Global-Pipelined (GP)* algorithm to get the top- k results which are ranked by the IR-style ranking strategy shown in Section 2.4. The aim of the algorithm is to find a proper order of generating JTTs in order to stop early before all the JTTs are generated. It employs the *priority preemptive, round robin* protocol [18] to find results from each query tuple set prefix in a pipelined way, thus each CN can avoid being fully evaluated.

For a keyword query Q , given a CN C , let the set of query tuple sets of C be $\{R_1^Q, R_2^Q, \dots, R_m^Q\}$. Tuples in each R_i^Q are sorted in non-increasing order of their scores computed by Eq. 2. Let $R_i^Q.t_j$ be the j -th tuple in R_i^Q . In each R_i^Q , we use $R_i^Q.cur$ to denote the current tuple such that the tuples before the position of the tuple are all processed, and we use $R_i^Q.cur \leftarrow R_i^Q.cur + 1$ to move $R_i^Q.cur$ to the next position. $q(t_1, t_2, \dots, t_m)$ (where t_i is a tuple, and $t_i \in R_i^Q$) denotes the parameterized query which checks whether the m tuples can form a valid JTT. For each tuple $R_i^Q.t_j$, we use $\overline{score}(C.R_i^Q.t_j, Q)$ to denote the upper bound score for all the JTTs of C that contain the tuple $R_i^Q.t_j$, defined as follows:

$$\overline{score}(C.R_i^Q.t_j, Q) = \frac{t_j.score + \sum_{1 \leq i' \leq m \wedge i' \neq i} C.R_{i'}^Q.t_1.score}{size(C)} \quad (3)$$

According to the tuple monotonicity property of Eq. (1) and the sorting order of tuples, among the unprocessed tuples of $C.R_i^Q$, $\overline{score}(C.R_i^Q.cur, Q)$ has the maximum value.

Algorithm GP initially mark all tuples in $C.R_i^Q$ ($1 \leq i \leq m$) of each CN C as un-processed except for the top-most ones. Then in each while iteration (one round), the un-processed tuple which maximizes the \overline{score} value is selected for processing. Suppose tuple $C_0.R_s^Q.cur$ maximizes \overline{score} , processing $C_0.R_s^Q.cur$ is done by joining it with the processed tuples in the other query tuple sets of C_0 to find valid JTTs: all the combinations as $(t_1, t_2, \dots, t_{s-1}, R_s^Q.cur, t_{s+1}, \dots, t_m)$ are tested, where t_i is a processed

tuple of $C_0.R_i^Q$ ($1 \leq i \leq m, i \neq s$). If the k -th relevance score of the found results is larger than \overline{score} values of all the un-processed tuples in all the CNs, it can stop and output the k found results with the largest relevance scores because no results with higher scores can be found in the further evaluation.

One drawback of the GP algorithm is that when a new tuple $C.R_i^Q.cur$ is processed, it tries all the combinations of processed tuples $(t_1, t_2, \dots, t_{s-1}, t_{s+1} \dots, t_m)$ to test whether each combination can be joined with $C.R_i^Q.cur$. This operation is costly due to extremely large number of combinations when the number of processed tuples becomes large [19]. SPARK [9] proposes the *Skyline-Sweeping* algorithm to reduce the number of combinations test. SPARK uses a priority queue \mathbb{Q} to keep the set of seen but not tested combinations ordered by the priority defined as the score of the hypothetical JTT corresponding to each combination. In each round, the combination in \mathbb{Q} with the maximum priority is tested, then all its adjacent combinations are inserted into \mathbb{Q} but only the combinations that have the high priorities are tested. SPARK still can not avoid testing a huge number of combinations which cannot produce results, though the number of combinations test is highly reduced compared to DISCOVER2.

This paper evaluates the CNs in a pipelined way like [4] and [9], but also employs the following two optimization strategies, whose high efficiencies are shown in [2,14,15]: (1) sharing the computational cost among CNs; and (2) adopting tuple reduction.

3.3 Keyword Search in Relational Data Streams

The most related projects to our paper are *S-KWS* [14] and *KDynamic* [20,15], which try to find new results or expired results for a given keyword query over an open-ended, high-speed large relational data stream [13]. They adopt the schema-based framework since the database is not static. This paper deals with a different problem from *S-KWS* and *KDynamic*, though all need to respond to continual queries in a dynamic environment. *S-KWS* and *KDynamic* focus on finding all query results. On the contrary, our methods maintain the top- k results, which is less sensitive to the updates of the underlying databases because not every new or expired results change the top- k results.

S-KWS maps each CN to a left-deep *operator tree*, where leaf operators (nodes) are tuple sets, and interior operators are joins. Then the operator trees of all the CNs are compacted into an *operator mesh* by collapsing their common subtrees. Joins in the operator mesh are evaluated in a bottom-to-top manner. A join operator has two inputs and is associated with an output buffer which saves its results (partial JTTs). The output buffer of a join operator becomes input to many other join operators that share the join operator. A new result that is newly outputted by a join operator will be a new arrival input to those joins sharing it. The operator mesh has two main shortcomings [19]: (1) only the left part of the operator trees can be shared; and (2) a large number of intermediate tuples, which are computed by many join operators in the mesh with high processing cost, will not be eventually output in the end.

For overcoming the above shortcomings of *S-KWS*, *KDynamic* formalizes each CN as a rooted tree, whose root is defined to be the node r such that the maximum path

from r to all leaf nodes of the CN is minimized; and then compresses all the rooted trees into a \mathcal{L} -Lattice by collapsing the common subtrees. Fig. 3(a) shows the lattice of two hypothetical CNs. Each node V in the Lattice is also associated with an output buffer, which contains the tuples in V that can join at least one tuple in the output buffer of its each child node. Thus, each tuple in the output buffer of each top-most node V , i.e., the root of a CN, can form JTTs with tuples in the output buffers of its descendants. The new JTTs involving a new tuple are found in a two-phase approach. In the filter phase, as illustrated in Fig. 3(b), when a new tuple t_{new} is inserted into node R_4 , $KDynamic$ uses selections and semi-joins to check if (1) t_{new} can join at least a tuple in the output buffer of each child node of R_4 ; and (2) t_{new} can join at least a tuple in the output buffers of the ancestors of R_4 . The new tuples that can not pass the checks are pruned; otherwise, in the join phase (shown in Fig. 3(c)), a joining process is initiated from each tuple in the output buffer of each root node that can join t_{new} , in a top-down manner, to find the JTTs involving t_{new} .

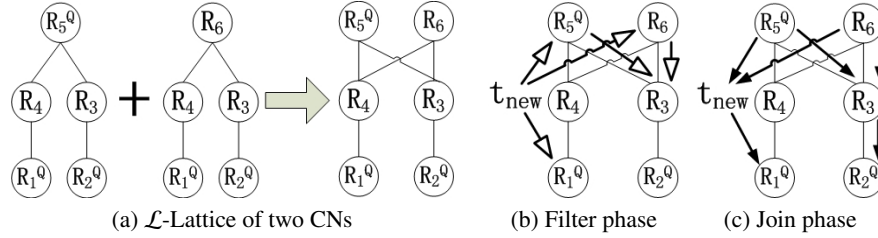


Fig. 3. Query processing in $KDynamic$

In this paper, we incorporate the ranking mechanisms and the pipelined evaluation into the query processing method of $KDynamic$ to support efficient top- k keyword search in relational databases.

4 Continual Top- k Keyword Search in Relational Databases

4.1 Overview

Database updates bring two orthogonal effects on the current top- k results:

1. They change the values of df_w , N , and $avdl$ in Eq. (2) and hence change the relevance scores of existing results.
2. New JTTs may be generated due to insertions. Existing top- k results may be expired due to deletions.

Although the second effect is more drastic, the first effect is not negligible for long-term database modifications. Thus, we can not neglect all the JTTs that are not the current top- k results because some of them have the potential of becoming the top- k results in the future. This paper solves this problem by bounding the future relevance score of each result. We use $score^u$ to denote the upper bound of relevance score for each result.

Then, the results whose $score^u$ values are not larger than relevance score of the top- k -th results can be safely ignored.

The second challenge is shortage of top- k results because they can be expired due to deletions. Since the value k is rather small compared to the huge number of all the valid JTTs, the possibility of deleting a top- k result is rather small. In addition, new top- k results can also be formed by new tuples. Thus, if the insertion rate is not much smaller than the deletion rate, the possibility of occurring of top- k results shortage would be small. However, this possibility would be high if the deletion rate is much larger, which can result in frequent top- k results refilling operations. It worth noting that the top- k results shortage can also be caused by the relevance score changing of results. Our solution to this problem is to compute the top- $(k + \Delta k)$ ($\Delta k > 0$) results instead of the necessary k . Δk is a margin value. Then, we can stand up to Δk times of deletion of top results when maintaining the top- k results. The setting of Δk is important. If Δk is too small, it may has a high possibility to refill. If Δk is too large, the efficiency of handling database modifications is decreased. Instead of analyzing the update behavior of the underlying database to estimate an appropriate Δk value, we enlarge Δk on each time of top- k results shortage until it reaches a value such that the occurring frequency of top- k results shortage falls below a threshold.

On the contrary, after maintaining the top- k results for a long time, the number of computed top results maybe larger than $(k + \Delta k)$, especially when the insertion rate is high. In such cases, the top- k results maintaining efficiency is decreased because we need to update the relevance scores for more results and join the new tuples with more tuples than necessary. As shown in the experimental results, such extra cost is not negligible for long-term database modifications. Therefore, we need to reverse the pipelined query evaluation if there are too many computed top results.

In brief, when a continual keyword query is registered, we first generate the set of CNs and compact them into a lattice \mathcal{L} . Then, the initial top- k results is found by processing tuples in \mathcal{L} in a pipelined way until the $score^u$ values of the un-seen JTTs are not larger than relevance score of the top- $(k + \Delta k)$ -th result (which is denoted by $\mathcal{L}.\theta$). When maintaining the top- k results, we only find the new results that are with $score^u > \mathcal{L}.\theta$. The pipelined evaluation of \mathcal{L} is resumed if the number of found results with $score^u > \mathcal{L}.\theta$ falls below k , or is reversed if the above number is larger than $(k + \Delta k)$. The method of computing $score^u$ for results is introduced in Section 4.2. Section 4.3 and Section 4.4 describe our method of computing the initial top- k results and maintaining the top- k results, respectively. Then, two techniques which can highly improve the query processing efficiency are presented in Section 4.5 and Section 4.6.

4.2 Computing Upper Bound of Relevance Scores

Let us recall the function for computing tuple scores given in Eq. (2):

$$tscore(t, Q) = \sum_{w \in t \cap Q} \frac{1 + \ln(1 + \ln(tf_{t,w}))}{1 - s + s \cdot \frac{dl_t}{avdl}} \cdot \ln\left(\frac{N}{df_w + 1}\right).$$

We assume that the future values of each $\ln\left(\frac{N}{df_w+1}\right)$ and $avdl$ both have an upper bound $\ln^u\left(\frac{N}{df_w+1}\right)$ and $avdl^u$, respectively. Then, we can derive the upper bound of the future tuple score for each tuple t as:

$$t.score^u = \sum_{w \in t \cap Q} \frac{1 + \ln(1 + \ln(df_{t,w}))}{1 - s + s \cdot \frac{dl_t}{avdl^u}} \cdot \ln^u\left(\frac{N}{df_w + 1}\right). \quad (4)$$

Hence, the upper bound of the future relevance score of a JTT T is:

$$T.score^u = \sum_{t \in T} t.score^u \cdot \frac{1}{size(T)}. \quad (5)$$

Note that the function in Eq. (5) also has the tuple monotonicity property on $t.score^u$.

On query registration, each $\ln^u\left(\frac{N}{df_w+1}\right)$ is computed as $\ln\left(\frac{N}{df_w(1-\Delta df_w)+1}\right)$, and each $avdl^u$ is computed as $avdl(1 + \Delta avdl)$, where Δdf_w and $\Delta avdl$ both are set as small values ($= 1\%$). When maintaining the top- k results, we continually monitor the change of statistics to determine whether all the $\ln\left(\frac{N}{df_w+1}\right)$ and $avdl$ values below their upper bounds. At each time that any $\ln\left(\frac{N}{df_w+1}\right)$ or $avdl$ value exceeds its upper bound, the Δdf_w or $\Delta avdl$ is enlarged until the frequencies of exceeding the upper bounds fall below a small number.

Example 3. Table 2 shows the $t.score^u$ values of the six matched tuples in Example 1 by setting $\Delta df_w = 20\%$ and $\Delta avdl = 10\%$. Hence, $T_1.score^u = 7.42$, $T_2.score^u = 4.23$ and $T_3.score^u = 3.88$.

Table 2. Upper bounds of tuple scores

Tuple	a_1	a_3	a_5	p_1	p_2	p_5
$t.score^u$	4.23	3.64	3.60	3.52	7.42	3.57

4.3 Finding Initial Top- k Results

Fig. 4 shows the \mathcal{L} -lattice of the seven CNs in Example 2. We use V_i to denote a node in \mathcal{L} . Particularly, V_i^Q denotes a lattice node of query tuple set, and $V_i^Q.R^Q$ denotes the query tuple set of V_i^Q . The dual edges between two nodes, for instance, V_1^Q and V_5 , indicate that V_5 is a dual child of V_1^Q . A node V_i in \mathcal{L} can belong to multiple CNs. We use $V_i.CN$ to denote the set of CNs that node V_i belongs to. For example, $V_8^Q.CN = \{CN_2, CN_3, CN_6, CN_7\}$. Tuples in each query tuple set $V_i^Q.R^Q$ are sorted in non-increasing order of $t.score^u$. We use $V_i^Q.cur$ to denote the current tuple such that the tuples before the position of the tuple are all processed, and we use $V_i^Q.cur \leftarrow V_i^Q.cur + 1$ to move $V_i^Q.cur$ to the next position. Initially, for each node V_i^Q in \mathcal{L} , $V_i^Q.cur$ is set as the top tuple in $V_i^Q.R^Q$. In Fig. 4, $V_i^Q.cur$ of the four nodes are denoted by arrows. For a node V_i that is of a free tuple set R_i^F , we regard all the tuples of R_i^F as its processed tuples for all the times. We use $V_i.output$ to indicate the output buffer of V_i , which contains its processed tuples that can join at least one tuple in the output buffer of each child node of V_i . Tuples in $V_i.output$ are also referred as the outputted tuples of V_i .

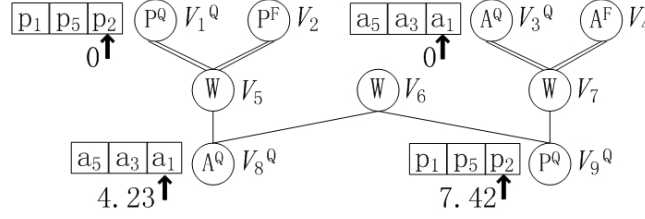


Fig. 4. The constructed lattice of the seven CNs in Example 2

In order to find the top- k results in a pipelined way, we need to bound the $score^u$ values of the un-found results. For each tuple t_j of $V_i^Q.R^Q$, the maximal $score^u$ values of JTTs that t_j can form is defined as follows:

$$\overline{score^u}(V_i^Q, t_j, Q) = \begin{cases} 0, & \text{a child node of } V_i^Q \text{ has empty output buffer,} \\ \max_{C \in V_i^Q.CN} (\overline{score^u}(C.R^Q.t_j, Q)), & \text{otherwise} \end{cases} \quad (6)$$

where $\overline{score^u}(C.R^Q.t_j, Q)$ indicates the maximal $score^u$ for all the JTTs of C that contain tuple t_j , and is obtained by replacing $tscore$ in Eq. (3) with $tscore^u$. If a child of V_i^Q has empty output buffer, processing any tuple at V_i^Q can not produce JTTs; hence $\overline{score^u}(V_i^Q, t_j, Q) = 0$ in such cases, which can choke the processing tuples at V_i^Q until all its child nodes have non-empty output buffers. According to Eq. (6) and the tuples sorting order, among the un-processed tuples of $V_i^Q.R^Q$, $\overline{score^u}(V_i^Q, V_i^Q.cur, Q)$ has the maximum value. We use $\overline{score^u}(V_i^Q, Q)$ to denote $\overline{score^u}(V_i^Q, V_i^Q.cur, Q)$. In Fig. 4, $\overline{score^u}(V_i^Q, Q)$ values of the four V_i^Q nodes are shown next to the arrows. For example, $\overline{score^u}(V_8^Q, Q) = \max_{C \in \{CN_2, CN_3, CN_6, CN_7\}} (\overline{score^u}(C.A^Q.a_1, Q)) = 4.23$.

Algorithm 1 outlines our pipelined algorithm of evaluating the lattice \mathcal{L} to find the initial top- k results, which is similar to the GP algorithm. Lines 1-3 are the initialization step to sort tuples in each query tuple set and to initialize each $V_i^Q.cur$. Then in each while iteration (lines 4-8), the un-processed tuple in all the V^Q nodes that maximizes $\overline{score^u}$ is selected to be processed. Processing the selected tuples is done by calling the procedure *Insert*. Algorithm 1 stops when $\max_{V_i^Q \in \mathcal{L}} \overline{score^u}(V_i^Q, Q)$ is not larger than the relevance score of the top- $(k + \Delta k)$ -th found results. The procedure *Insert*(V_i, t) is provided in *KDynamic*, which updates the output buffers for V_i (line 13) and all its ancestors (lines 17-18), and finds all the JTTs containing tuple t by calling the procedure *EvalPath* (line 16). We will explain procedure *Insert* using examples later. The recursive procedure *EvalPath*($V_i, t, path$) is provided in *KDynamic* too, which constructs JTTs using the outputted tuples of V_i 's descendants that can join t . The stack *path*, which records where the join sequence comes from, is used to reduce the join cost.

Example 4. In the first round, tuple $V_9^Q.p_2$ is processed by calling *Insert*(V_9^Q, p_2). Since V_9^Q is the root node of CN_1 , *EvalPath* is called and JTT $T_1 = p_2$ is found. Then, for the two father nodes of V_9^Q , V_6 and V_7 , $V_6.output$ is not updated because $V_8^Q.output = \emptyset$, $V_7.output$ is updated to $\{w_1, w_7\}$ because p_2 can join w_1 and w_7 . And then, for the two father nodes of V_7 , V_3^Q and V_4 , $V_3^Q.output$ is not updated since V_3^Q has no processed

Algorithm 1: *EvalStatic-Pipelined* (lattice \mathcal{L} , the top- k value k , Δk)

```

1   $topk \leftarrow \emptyset$ : the priority queue for storing found JTTs ordered by score;
2  Sort tuples of each  $V_i^Q.R^Q$  in non-increasing order of  $tscore^u$ ;
3  foreach node  $V_i^Q$  in  $\mathcal{L}$  do let  $V_i^Q.cur \leftarrow V_i^Q.R^Q.t_1$ ;
4  while  $\max_{V_i^Q \in \mathcal{L}} \overline{score^u}(V_i^Q, Q) > topk[k + \Delta k].score$  do
5      Suppose  $\overline{score^u}(V_0^Q, Q) = \max_{V_i^Q \in \mathcal{L}} \overline{score^u}(V_i^Q, Q)$ ;
6       $path \leftarrow \emptyset$ ; // A stack which records the join sequence
7       $Insert(V_0^Q, V_0^Q.cur)$ ; // Processing tuple  $V_0^Q.cur$  at  $V_0^Q$ 
8       $V_0^Q.cur \leftarrow V_0^Q.cur + 1$ ;
9  Output the first  $k$  results in  $topk$ ;
10  $\mathcal{L}.t \leftarrow topk[k + \Delta k].score$ ;

11 Procedure Insert(lattice node  $V_i$ , tuple  $t$ )
12 if  $t \notin V_i.output$  and  $t$  can join at least one outputted tuple of every child of  $V_i$  then
13     Insert  $t$  into  $V_i.output$ ;
14 if  $t \in V_i.output$  then
15     Push  $(V_i, t)$  to  $path$ ;
16     if  $V_i$  is a root node then  $topk \leftarrow topk \cup EvalPath(V, t, path)$ ;
17     foreach father node of  $V_i$ ,  $V_{i'}$  in  $\mathcal{L}$  do
18         foreach tuple  $t'$  belongs to  $V_{i'}$  that can join  $t$  do  $Insert(V_{i'}, t')$ ;
19     Pop  $(V_i, t)$  from  $path$ ;

20 Procedure EvalPath(lattice node  $V_i$ , tuple  $t$ , stack  $path$ )
21  $\mathcal{T} \leftarrow \{t\}$ ; // The set of found JTTs
22 foreach child node of  $V_i$ ,  $V_{i'}$  in  $\mathcal{L}$  do
23      $\mathcal{T}' \leftarrow \emptyset$ ; // The set of JTTs that rooted at tuples of node  $V_{i'}$ 
24     if  $V_{i'} \in path$  then
25         let  $t'$  be the tuple of node  $V_{i'}$  that is stored in  $path$ ;
26          $\mathcal{T}' \leftarrow EvalPath(V_{i'}, t', path)$ ;
27     else
28         foreach tuple  $t' \in V_{i'}.output$  that join  $t$  do
29              $\mathcal{T}' \leftarrow \mathcal{T}' \cup EvalPath(V_{i'}, t', path)$ ; // Union the JTTs that rooted
                at different tuples of  $V_{i'}$ 
30      $\mathcal{T} \leftarrow \mathcal{T} \times \mathcal{T}'$ ; // Compute the Cartesian Product
31 return  $\mathcal{T}$ ;

```

tuples, $V_4.output$ is set as $\{a_2\}$ because there is only one tuple a_2 in A^F that can join w_1 and w_7 . Since V_4 is the root node (of CN_5), $EvalPath(V_4, a_2, path)$ is called but no results are found because the only one found JTT $p_2 \leftarrow w_7 \rightarrow a_2 \leftarrow w_7 \rightarrow p_2$ is not a valid result. After processing tuple $V_9^Q.p_2$, $\overline{score^u}(V_3^Q, Q) = 3.82$ and $\overline{score^u}(V_9^Q, Q) = 3.57$. In the second round, tuple $V_8^Q.a_1$ is processed, which finds results $T_2 = a_1$ and $T_3 = p_2 \leftarrow w_1 \rightarrow a_1$. Then, $V_2.output = \{p_4\}$, $V_5.output = \{w_1, w_4\}$, $V_6.output = \{w_1\}$, $\overline{score^u}(V_1^Q, Q) = 3.18$, and $\overline{score^u}(V_8^Q, Q) = \overline{score^u}(CN_3.A^Q.a_3, Q) = 3.69$. In the third-fifth rounds, tuples $V_3^Q.a_1$, $V_3^Q.a_3$ and $V_3^Q.a_5$ are processed, which insert a_1 into $V_3^Q.output$ and no results found. In the sixth round, tuple $V_8^Q.a_3$ is processed, which finds results a_3 and $a_1 \leftarrow w_4 \rightarrow p_4 \leftarrow w_6 \rightarrow a_3$. Then, Algorithm 1 stops because the relevance score of the third result in the queue $topk$ (suppose $\Delta k = 0$) is larger than all the $\overline{score^u}(V_i^Q, Q)$ values. Fig. 5 shows the snapshot of \mathcal{L} after finding the top-3 results. Thus, $\theta = 3.68$ after the evaluation.

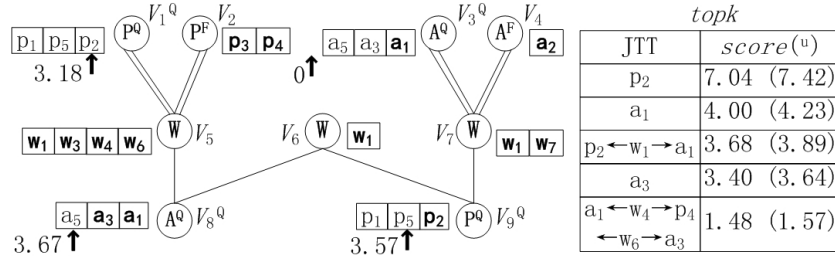


Fig. 5. After finding the top-3 results (tuples in the output buffers are shown in bold)

After the execution of Algorithm 1, $\overline{score^u}$ values of all the un-found results are not larger than $\mathcal{L}.\theta$. Results in the queue $topk$ can be categorized into three kinds. The first kind are the $(k + \Delta k)$ results that are with $\overline{score^u} \geq \mathcal{L}.\theta$, which are the initial top- $(k + \Delta k)$ results. The second kind are with $\overline{score} < \mathcal{L}.\theta$ and $\overline{score^u} \geq \mathcal{L}.\theta$, which are called the *potential* top- $(k + \Delta k)$ results because they have the potential to become the top- $(k + \Delta k)$ results. The third kind are with $\overline{score^u} \leq \mathcal{L}.\theta$. As shown in the experiment, the results of the last kind may have a large number. However, we can not discard them because some of them may become the first two kinds when maintaining the top- k results.

4.4 Maintaining Top- k Results

Algorithm 2 shows our algorithm of maintaining top- k results. A database update operator is denoted by $OP(t, R_t)$, which represents a tuple t of relation R_t is inserted (if OP is a insertion) or deleted (if OP is a deletion). Note that the database updates is modeled as deletions followed by insertions. For a new arrival $OP(t, R_t)$, Algorithm 2 first checks whether the $\ln(\frac{N}{df_w+1})$ and $avdl$ values of relation R_t exceed their upper bounds. If some $\ln(\frac{N}{df_w+1})(s)$ or $avdl$ exceeds their upper bounds, we enlarge² the corresponding $\Delta df_w(s)$

² The methods of enlarging Δdf_w , $\Delta avdl$ and Δk are introduced in detail in the experiments.

or $\Delta avdl$ (line 3), and then update the $score$ and $score^u$ values for all the tuples in R_i^Q and all the results in the queue $topk$ using the enlarged $\ln\left(\frac{N}{df_w+1}\right)(s)$ or $avdl$ (line 4); otherwise, we update the relevance scores for the results in $topk$ that are with $score^u \geq \mathcal{L}.\theta$ (line 6). Then, we insert t into \mathcal{L} to find the new results if OP is an insertion (lines 7-13), or delete the expired JTTs and t from \mathcal{L} if OP is a deletion (lines 14-17). Lines 7-17 are explained in detail latter. And then, the $\overline{score^u}(V_i^Q, Q)$ of some nodes may be large than $\mathcal{L}.\theta$, which can be caused by three reasons: (1) the upper bound scores of tuples of relation R_i are increased; (2) the $\overline{score^u}(V_i^Q, Q)$ of some nodes are increased from 0 after inserting the new tuple into \mathcal{L} ; and (3) new CNs are added into \mathcal{L} . Therefore, in lines 18-19, we process tuples using procedure *Insert* until all the $\overline{score^u}(V_i^Q, Q)$ values are not larger than $\mathcal{L}.\theta$.

Algorithm 2: *Maintain*(the evaluated lattice \mathcal{L} , the top- k value k , Δk)

```

1 while a new database modification  $OP(t, R_i)$  arrives do
2   if Some  $\ln\left(\frac{N}{df_w+1}\right)$  (or  $avdl$ ) exceed their upper bounds after applying  $OP$  then
3     Enlarge the corresponding  $\Delta df_w$  (or  $\Delta avdl$ ) value(s);
4     Update relevance scores for tuples in  $R_i^Q$  and results in  $topk$ ;
5   else
6     Update  $score$  for results in  $topk$  that are with  $score^u \geq \mathcal{L}.\theta$ ;
7   if  $OP$  is an insertion then                                     // Insert  $t$  into  $\mathcal{L}$ 
8     if  $t$  is an un-matched tuple then
9       foreach node  $V_i$  in  $\mathcal{L}$  that of  $R_i^F$  do  $Insert(V_i, t)$ ;
10    else
11      if  $R_i^Q$  is new then add the new CNs into  $\mathcal{L}$ ;
12      Insert  $t$  into  $R_i^Q$  in descending order of  $tscore^u$ ;
13      foreach  $V_i^Q$  that of  $R_i^Q$  and has  $\overline{score^u}(V_i^Q, t, Q) > \mathcal{L}.\theta$  do  $Insert(V_i^Q, t)$ ;
14    else if  $OP$  is a deletion then                                   // Delete  $t$  from  $\mathcal{L}$ 
15      Delete the results that contain  $t$  and are with  $score^u \geq \mathcal{L}.\theta$  from  $topk$ ;
16      if  $t$  is a matched tuple then remove  $t$  from  $R_i^Q$ ;
17      foreach node  $V_i$  in  $\mathcal{L}$  such that  $t \in V_i.output$  do  $Delete(V_i, t)$ ;
18    while  $\max_{V_i^Q \in \mathcal{L}} \overline{score^u}(V_i^Q, Q) > \mathcal{L}.\theta$  do
19      foreach node  $V_i^Q$  that is with  $\overline{score^u}(V_i^Q, Q) > \mathcal{L}.\theta$  do  $Insert(V_i^Q, V_i^Q.cur)$ ;
20    if  $|\{T | T \in topk, T.score \geq \mathcal{L}.\theta\}| < k$  then                // Resume the evaluation of  $\mathcal{L}$ 
21      Enlarge  $\Delta k$  and then resume the execution of EvalStatic-Pipelined;
22    else if  $|\{T | T \in topk, T.score \geq \mathcal{L}.\theta\}| > (k + \Delta k)$  then
23       $RollBack(\mathcal{L}, k, \Delta k)$ ;                                       // Reverse the evaluation of  $\mathcal{L}$ 
24      Report the new first  $k$  results in  $topk$  if they are changed;

25 Procedure  $Delete(V_i, t)$ 
26 Delete  $t$  from  $V_i.output$ ;
27 foreach father node of  $V_i, V_{i'}$  in  $\mathcal{L}$  do
28   foreach tuple  $t'$  in  $V_{i'}.output$  that can join  $t$  only do
29      $Delete(V_{i'}, t')$ ;                                           // Call Delete recursively

```

Finally, in lines 20-23, we count the number of results that are with $score^u \geq \mathcal{L}.\theta$. If the number is smaller than k , Δk is enlarged, and then the *EvalStatic-Pipelined* algorithm (without the initialization step) is called to further evaluate \mathcal{L} . If the number is larger than $k + \Delta k$, the algorithm *RollBack*, which is described at the end of this subsection, is called to rollback the evaluation of \mathcal{L} . In any case, at the end of handling the *OP*, we have $\max_{V_i^Q \in \mathcal{L}} \overline{score^u}(V_i^Q.t_{cur}, Q) \leq \text{topk}[k].score$. Therefore, the k results in *topk* that have the largest relevance scores are the top- k results. We do not process the results in *topk* that are with $score^u \leq \mathcal{L}.\theta$ in line 6 and line 15, because they can have a large number and do not have the potential to become top- k results. However, after the execution of lines 4 and 21, $score^u$ of some of them may become larger than $\mathcal{L}.\theta$, because their $score^u$ values may be enlarged in line 4 and the $\mathcal{L}.\theta$ may be decreased in line 21. Therefore, all the results in *topk* need to be considered in lines 4 and 21. Note that we have to firstly check whether some of them have expired due to deletions.

In lines 7-13, the new tuple t is processed differently according to whether it contains the keywords. If t is an un-matched tuple, it is inserted into each node of R_t^F using the procedure *Insert* (line 9). If t is a matched tuple, inserting it into \mathcal{L} is more complicated. First, if t introduces a new non-empty query tuple set R_t^Q , we add the new CNs involving R_t^Q into the lattice. Fig. 6 illustrates the process of inserting a new CN into the lattice shown in Fig. 5. Assuming that $W \rightarrow P^Q$ is the largest common subtree of the new CN and \mathcal{L} , and V_f is the father node of $W \rightarrow P^Q$ in the new CN, then the new CN is added by setting V_7 as the child of V_f . If V_f is a free tuple set and it does not have other child nodes as shown in Fig. 6, *Insert*(V_f, t') is called for each tuple t' of V_f that can join tuples in $V_7.output$. Further evaluation at the nodes of the new CN, if necessary, will be done in lines 18-19. Second, t is added into the query tuple set R^Q (line 12), and then for each node V_i^Q of R_t^Q , *Insert*(V_i^Q, t) is called when $\overline{score^u}(V_i^Q.R^Q.t, Q) > \mathcal{L}.\theta$ (line 13), i.e., t has the potential to form JTTs that are with $score^u > \mathcal{L}.\theta$.

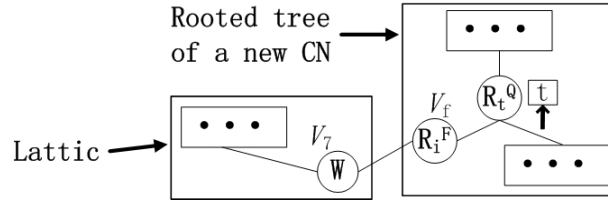


Fig. 6. Inserting a new CN into the lattice

If *OP* is a deletion, for each node V_i in \mathcal{L} such that $t \in V_i.output$, we delete t from $V_i.output$ using the procedure *Delete*, which is provided by *KDynamic*. Procedure *Delete* first removes t from $V_i.output$, and then checks whether some outputted tuples of the ancestors of V_i need to be removed (lines 27-29). For instance, if the tuple a_3 is deleted from the lattice node V_8^Q shown in Fig. 5, tuples w_3 and w_6 are deleted from $V_5.output$ too because they can join a_3 only, among tuples in $V_8^Q.output$.

Algorithm 3 outlines out algorithm to reverse the execution of the pipelined evaluation of the lattice. In the beginning, $\mathcal{L}.\theta$ is set as the relevance score of the $(k + \Delta k)$ -th

result in the queue *topk* (line 1). Then, the processing on each processed tuple $t \in R_i^Q$ that is of $\overline{score^u}(V_i^Q.R^Q.t, Q) \leq \mathcal{L}.\theta$ is reversed (lines 4-6). We use $V_i^Q.cur - 1$ to denote the tuple just before $V_i^Q.cur$. If $t \in R_i^Q.output$, the results involving by t are firstly deleted from *topk*, and then t is deleted from $V_i^Q.output$ by calling the procedure *Delete*.

Algorithm 3: *RollBack*(a lattice \mathcal{L} , the top-k value k , Δk)

```

1  $\mathcal{L}.\theta \leftarrow topk[k + \Delta k].score;$ 
2 foreach node  $V_i^Q$  in  $\mathcal{L}$  do
3   while  $\overline{score^u}(V_i^Q.cur - 1, Q) \leq \mathcal{L}.\theta$  do
4     if  $V_i^Q.cur - 1 \in V_i^Q.output$  then
5       Remove the results that are of CNs in  $V_i^Q.CN$  and contain tuple  $V_i^Q.cur - 1$ 
        from topk;
6       Delete( $V_i^Q, V_i^Q.cur - 1$ );           // Delete from the output buffer
7        $V_i^Q.cur \leftarrow V_i^Q.cur - 1;$ 

```

4.5 Caching Joined Tuples

In Algorithm 1 and Algorithm 2, procedure *Insert* and *Delete* may be called by multiple times upon multiple nodes for the same tuple. The core of the two procedures are the *select operations* (or semi-joins [15]). For example, in line 12 and line 18 of procedure *Insert*, we need to select the tuples that can join t from the output buffer of each child node of V_i and the set of processed tuples of each father node of V_i , respectively. Although such select operations can be done efficiently by the DBMS using indexes, the cost of handling t is high due to the large number of database accesses. For example, in our experiments, for a new tuple t , the maximal number of database accesses can be up to several hundred.

These select operations done for the same tuple t can be done efficiently by sharing the computational cost among them. Assume a new tuple w_0 is inserted into the lattice shown in Fig. 5, then procedure *Insert* is called by three times (*Insert*(V_5, w_0), *Insert*(V_7, w_0) and *Insert*(V_6, w_0)) and at most eight selections are done. All the eight select operations can be expressed using following two relational algebra expressions: $\pi_{aid}(\sigma_{wid=w_0}(W) \bowtie \sigma_{aid \in \mathcal{A}_i}(A))$ and $\pi_{pid}(\sigma_{w=w_0}(W) \bowtie \sigma_{p \in \mathcal{P}_j}(P))$, where \mathcal{A}_i and \mathcal{P}_j represent the set of tuples in the output buffer of a node or the set of processed tuples of a node. Since \mathcal{A}_i and \mathcal{P}_j can be different from each other, the eight select operations need to be evaluated individually. However, if we rewrite the above expressions as $\pi_{aid}(\sigma_{aid \in \mathcal{A}_i}(\sigma_{wid=w_0}(W) \bowtie (A)))$ and $\pi_{pid}(\sigma_{p \in \mathcal{P}_j}(\sigma_{w=w_0}(W) \bowtie (P)))$, the eight select operations would have two common sub-operations: $\sigma_{w=w_0}(W) \bowtie (A)$ and $\sigma_{w=w_0}(W) \bowtie (P)$. If the results of the two common sub-operations can be shared and do selections $\sigma_{aid \in \mathcal{A}_i}$ and $\sigma_{pid \in \mathcal{P}_j}$ in the main memory, the eight select operations can be evaluated involving only two database accesses.

Algorithm 4: *CanJoinOneOutputTuple*(lattice node V_i , tuple t)

```

1 Let  $R_i$  be the relation corresponding to the tuple set of  $V_i$ ;
2 if the tuples of relation  $R_i$  that can join  $t$  have not been stored then
3   | Query the tuples of relation  $R_i$  that can join  $t$  and store them;
4 foreach tuple  $t'$  of the stored tuples of relation  $R_i$  that can join  $t$  do
5   | if can find  $t'$  in  $V_i.output$  then return true;
6 return false;

```

Algorithm 4 shows our procedure to check whether tuple t can join at least one tuple in the output buffer of a lattice node V_i , which is called in line 12 of procedure *Insert*. In line 3, all the tuples in relation R_i that can join t are queried and cached in the main memory. This set of cached joined tuples can be reused every time when they are queried. The procedures for the select operations in line 18 of *Insert* and line 28 of *Delete* are also designed in this pattern, which are omitted due to the space limitation. Note that when the two procedures *Insert* and *Delete* are called recursively, select operations done in the above lines are also evaluated by these procedures. Therefore, for each tuple t , a tree of tuples, which is rooted at t and consist of all the tuples than can join t , is created. The tree of tuples can be seen as the cached localization information of t . It is created on-the-fly, i.e., along with the execution of procedures *Insert* and *Delete*, and its depth is determined by the recursion depth of the two procedures. The maximum recursion depth of procedures *Insert* and *Delete* is $\frac{CN_{\max}}{2} + 1$ [15], where CN_{\max} indicates the maximum size of the generated CNs. Hence, the height of this tree of tuples is bounded by $\frac{CN_{\max}}{2} + 1$ too.

Suppose a new tuple p_0 of P^Q is inserted into the two nodes of P^Q in the lattice shown in Fig. 5, Fig. 7 illustrates the select operations done in the procedure *Insert* (denoted as arrows in the left part) and the cached joined tuples of p_0 (shown in the right part). For instance, the arrows from V_9^Q to V_7 selects the tuples in relation W that can join p_0 . The three select operations are denoted by dashed arrows because they would not be done if results of the two select operations, from V_9^Q to V_7 and from V_9^Q to V_6 , are empty. For the same reason, the stored tuples of relation A that can join p_0 are denoted using dashed rectangles.

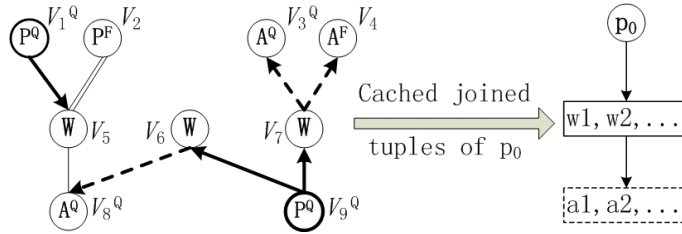


Fig. 7. Selections done in *Insert* and the cached joined tuples for a tuple p_0 of P^Q

When computing the initial top- k results, the database is static; hence the cached joined tuples of each tuple unchange and can be reused before the database is updated.

When maintain the top- k results, although the database is continually updated, we can assume the database unchange before t is handled. However, the cached joined tuples of t is expired after t is handled by Algorithm 2. As shown in the experimental results, caching the joined tuples can highly improve the efficiency of computing the initial top- k results and maintaining the top- k results.

4.6 Candidate Network Clustering

According to Eq. (3), $\overline{score^u}$ values of tuples in different CNs have great differences. For example, $\overline{score^u}$ values of tuples in CN_5 and CN_7 are smaller than that of tuples in CN_3 due to the large CN size. In algorithm GP, no tuples or only a small portion are joined in the CNs whose tuples have small \overline{score} values. If the CNs in Example 2 are evaluated by algorithm GP, A^Q of CN_7 and P^Q of CN_5 would have no processed tuples. However, in the lattice, a node R_i^Q can be shared by multiple CNs. Thus, when inserting a tuple t into R_i^Q , t is processed in all the CNs in $R_i^Q.CN$. As shown in Fig. 5, since V_8^Q is shared by CN_2 , CN_3 , CN_6 and CN_7 in the lattice, tuples a_1 and a_3 are processed in all these four CNs when processing them at V_8^Q , which results in *un-needed* operations at nodes V_2 and V_5 two un-needed results a_3 and $a_1 \leftarrow w_4 \rightarrow p_4 \leftarrow w_6 \rightarrow a_3$. We call the operations at V_2 and V_5 and the two JTTs as un-needed because they would not occur or be found if the CNs are evaluated separately. These un-needed operations can cause further un-needed operations when maintaining the top- k results. For example, we have to join a new unmatched tuple of relation P with four tuples in $V_5.output$.

The essence of the above problem is that CNs have different potentials in producing top- k results, and then the same tuple set can have different numbers of processed tuples in different CNs if they are evaluated separately. In order to avoid finding the un-needed results, the optimal method is merely to share the tuple sets that have the same number of processed tuples among CNs when they are evaluated separately. However, we cannot get these numbers without evaluating the CNs. As an alternative, we attempt to estimate this number for the tuple sets of each CN C according to following heuristic rules:

- If $Max(C) = \frac{\sum_{1 \leq i \leq m} C.R_i^Q.t_1.tscore^u}{size(C)}$, which indicates the maximum $score^u$ of JTTs that C can produce, is high, tuple sets of C have more processed tuples.
- If two CNs have the same $Max(C)$ values, tuple sets of the CN with larger size have more processed tuples.

Therefore, we can cluster the CNs using their $Max(C) \cdot \ln(size(C))$ values, where $\ln(size(C))$ is used to normalize the effect of CN sizes. Then, when constructing the lattice, only the subtrees of CNs in the same cluster can be collapsed. For example, $Max(C) \cdot \ln(size(C))$ values of the seven CNs of Example 2 are: 5.15, 2.93, 5.39, 6.84, 5.32, 5.70 and 3.03; hence they can be clustered into two clusters: $\{CN_2, CN_7\}$ and $\{CN_1, CN_3, CN_4, CN_5, CN_6\}$. Fig. 8 shows the lattice after finding the top-3 results if the CNs are clustered, where the three un-needed JTTs in Fig. 5 can be avoided. As shown in the experimental section, clustering the CNs can highly improve the efficiency in computing the initial top- k results and handling the database updates.

We cluster the CNs using the K -mean clustering algorithm [21], which needs an input parameter to indicate the number of expected clusters. We use $Kmean$ to indicate the ratio of this input parameter to the number of CNs. The value of $Kmean$ represents the trade-off between sharing the computation cost among CNs and considering their different potentials in producing top- k results. When $Kmean = 0$, the CNs is not clustered, then the CNs share the computation cost at the maximum extent. When $Kmean = 1$, all the CNs are evaluated separately. In our experiments, we find that $Kmean = 0.6$ is optimal both for computing the initial top- k results and handling the database updates.

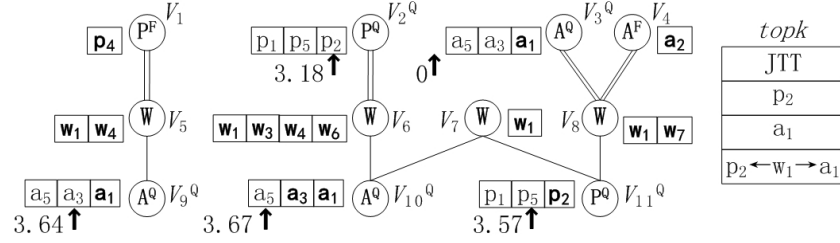


Fig. 8. After finding the top-3 results if the CNs are clustered into two clusters

5 Experimental Study

We conducted extensive experiments to test the efficiency of our methods. We use the DBLP dataset³. Note that DBLP is not continuously growing and is updated on a monthly basis. The reason we use DBLP to simulate a continuously growing relational dataset is because there is no real growing relational datasets in public, and many studies [4,9] on top- k keyword queries over relational databases use DBLP. The downloaded XML file is decomposed into relations according to the schema shown in Fig. 9. The two arrows from *PaperCite* to *Papers* denote the foreign-key-references from *paperID* to *paperID* and *citedPaperID* to *paperID*, respectively. The DBMS used is MySQL (v5.1.44) with the default “Dedicated MySQL Server Machine” configuration. All the relations use the MyISAM storage engine. Indexes are built for all primary key and foreign key attributes, and full-text indexes are built for all text attributes. All the algorithms are implemented in C++. We conducted all the experiments on a 2.53 GHz CPU and 4 GB memory PC running Windows 7.

5.1 Parameters

We use the following five parameters in the experiments: (1) k : the top- k value; (2) l : the number of keywords in a query; (3) IDF : the ratio of the number of matched tuples to the number of total tuples, i.e., $\frac{d_{fw}}{N}$; (4) CN_{max} : the maximum size of the generated CNs; and (5) $Kmean$: the ratio of the number of clusters of CNs to the number of CNs.

³ <http://dblp.mpi-inf.mpg.de/dblp-mirror/index.php/>

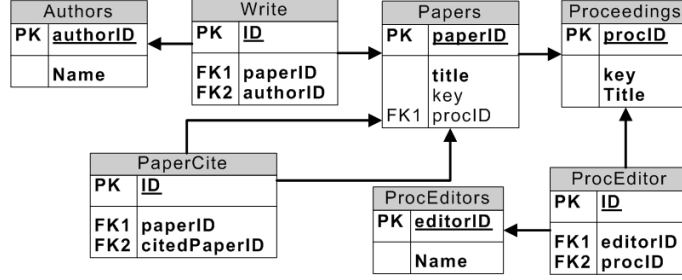


Fig. 9. The DBLP schema (PK stands for primary key, FK for foreign key)

The parameters with their default values (bold) are shown in Table. 3. The keywords selected are listed in Table. 4 with their IDF values, where the keywords in bold fonts are keywords popular in author names. Ten queries are constructed for every IDF value, each of which contains three selected keywords. For each l value, ten queries are constructed by selecting l keywords from the row of $IDF = 0.013$ in Table. 4. To avoid generating a small number of CNs for each query, one author name keyword of each IDF value always be selected for each query.

When k grows, the cost of computing the initial top- k results increases since we need to compute more results, and the cost of maintaining the top- k results also increases since there are more tuples in the output buffers of the lattice nodes. The parameter CN_{\max} has a great impact on keyword query processing because the number of generated CNs increases exponentially while CN_{\max} increases. And the number of matched tuples increases as IDF and l increase. Hence, the first four parameters k , l , IDF and CN_{\max} have effects on the scalability of our method.

Table 3. Parameters

Name	Values
k	50, 100 , 150, 200
l	2, 3 , 4, 5
IDF	0.003, 0.007, 0.013 , 0.03
CN_{\max}	4, 5, 6 , 7
$Kmean$	0, 0.20, 0.40, 0.60 , 0.80, 1

Table 4. Keywords and their IDF values

Keywords	IDF
ATM, embedded, navigation, privacy, scalable, Spatial, XML, Charles, Eric	0.004
clustering, fuzzy, genetic, machine, optimal, retrieval, sensor, semantic, video, James, Zhang	0.007
adaptive, architecture, database, evaluation, mobile, oriented, security, simulation, wireless, John, Wang	0.013
algorithm, design, information, learning, network, software, time, David, Michael	0.03

5.2 Exp-1: Initial Top- k Results Computation

In this experiment, we want to study the effects of the five parameters on computing the initial top- k results. We retrieve the data in the XML file sequentially until number of tuples in the relations reach the numbers shown in Table. 5. Then we run the algorithm *EvalStatic-Pipelined* on different values of each parameter while keeping the other four parameters in their default values. We use two measures to evaluate the effects of the parameters. The first is $\#R$, the number of found results in the queue *topk*. The second measure is T , the time cost of running the algorithm. Ten top- k queries are selected

for each combinations of parameters, and the average values of the metrics of them are reported in the following. In this experiment, Δdf_w ($= 1\%$), $\Delta avdl$ ($= 1\%$) and Δk ($= 1$) all have very small values because they will be enlarged adaptively when maintaining the top- k results.

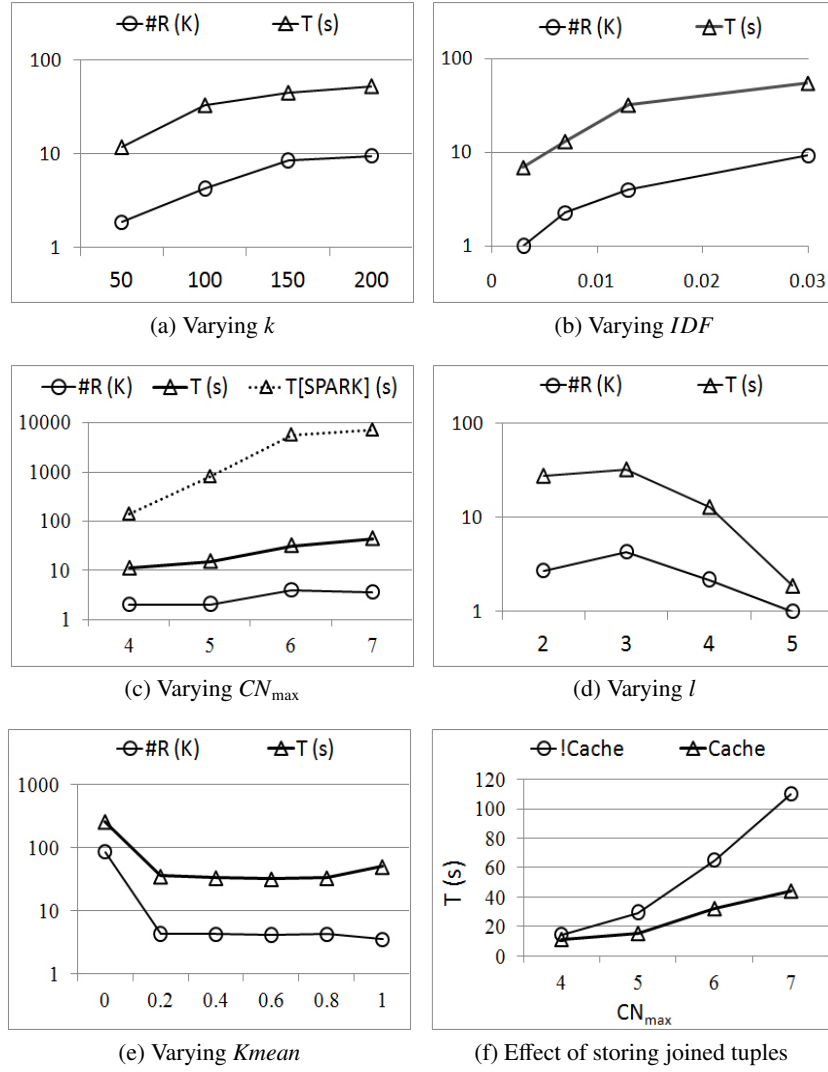
The main results of this experiment are given in Fig. 10. Note that the units for the y -axis are different for the three measures. Fig. 10(a), (b) and (c) show that the two measures all increases as k , idf and CN_{\max} grow. However, they do not show rapid increase in Fig. 10(a), (b) and (c), which imply the good scalability of our method. On the contrary, we can find rapid increase while CN_{\max} grows from the time cost of the method of [9] in finding the top- k results, which is shown in Fig. 10(c) and are denoted by $T[SPARK]$. Fig. 10(c) presents that, compared to the existing method, algorithm *EvalStatic-Pipelined* is very efficient in finding the top- k results. The reason is that evaluating the CNs using the lattice can achieve full reduction because all the tuples in the output buffer of the root nodes can form JTTs and can save the computation cost by sharing the common sub-expressions [15]. Fig. 10(d) shows that the effect of l seems more complicated: all the two measures may decrease when l increases. As shown in Fig. 10(d), $\#R$ and T even both achieve the minimum values when $l = 5$. This is because the probability that the keywords to co-appear in a tuple and the matched tuples can join is high when the number of keywords is large. Therefore, there are more JTTs that have high relevance scores, which results in larger θ and small values of the two measures.

Table 5. Tuple numbers of relations

Papers	PaperCite	Write	Authors	Proceedings	ProcEditors	ProcEditor
157,300	9,155	400,706	190,615	2,886	1,936	1,411

Fig. 10(e) presents the changing of the two measures when $Kmean$ varies. Since the results of the K -means clustering may be affected by the starting condition [21], for each $Kmean$ value, we run Algorithm 1 for 5 times on different starting condition for each keyword query and report the average experimental results. Note that the algorithm *EvalStatic* in *KDynamic* corresponding to $Kmean = 0$ since there is no CN clustering in *KDynamic*. From Fig. 10(e), we can find that clustering the CNs can highly improve the efficiency of computing the top- k results and the time cost decreases as $Kmean$ increases. However, when $Kmean = 1$, which indicates that all the CNs are evaluated separately, the time cost grows to a higher value than that when $Kmean$ is 0.6 or 0.8. Therefore, it is important to select a proper $Kmean$ value. The minimum T in this experiment is achieved on $Kmean = 0.6$; hence the default value of $Kmean$ is 0.6 in our experiments. As can be seen in the next section, $Kmean = 0.6$ also results in the minimum time cost of handling database modifications.

Fig. 10(f) compares the time cost of our method in finding the top- k results with that of *KDynamic*, while varying CN_{\max} . The time cost of *KDynamic* is denoted by “!Cache” because it does not cache the joined tuples for each tuple. We can find that caching the joined tuples for each tuple highly improves the efficiency of computing the top- k results. More important, the improvement increases as CN_{\max} grows. This is because when CN_{\max} grows, the times of calling the procedure *Insert* on each tuple

**Fig. 10.** Experimental results of calculating the initial top- k results

increases fast since the number of lattice nodes increases exponentially; hence the saved cost due to storing the joined tuples of each tuple grows as CN_{\max} grows.

From the curves of $\#R$ in Fig. 10, we can find that $\#R$ values are large in all the settings: about several thousand. Recall that *topk* contains three kinds of results. The number of the first kind of results is $k + \Delta k$, which is small compared to the $\#R$ values. Since $\Delta df_w (= 1\%)$, $\Delta avdl$ and Δk all have very small values, the number of potential top- $(k + \Delta k)$ results in *topk* is very small (< 10). Therefore, the third kind of results, which are with $score^u < \mathcal{L}.\theta$, is in the majority and has a larger number.

5.3 Exp-2: Top- k Result Maintenance

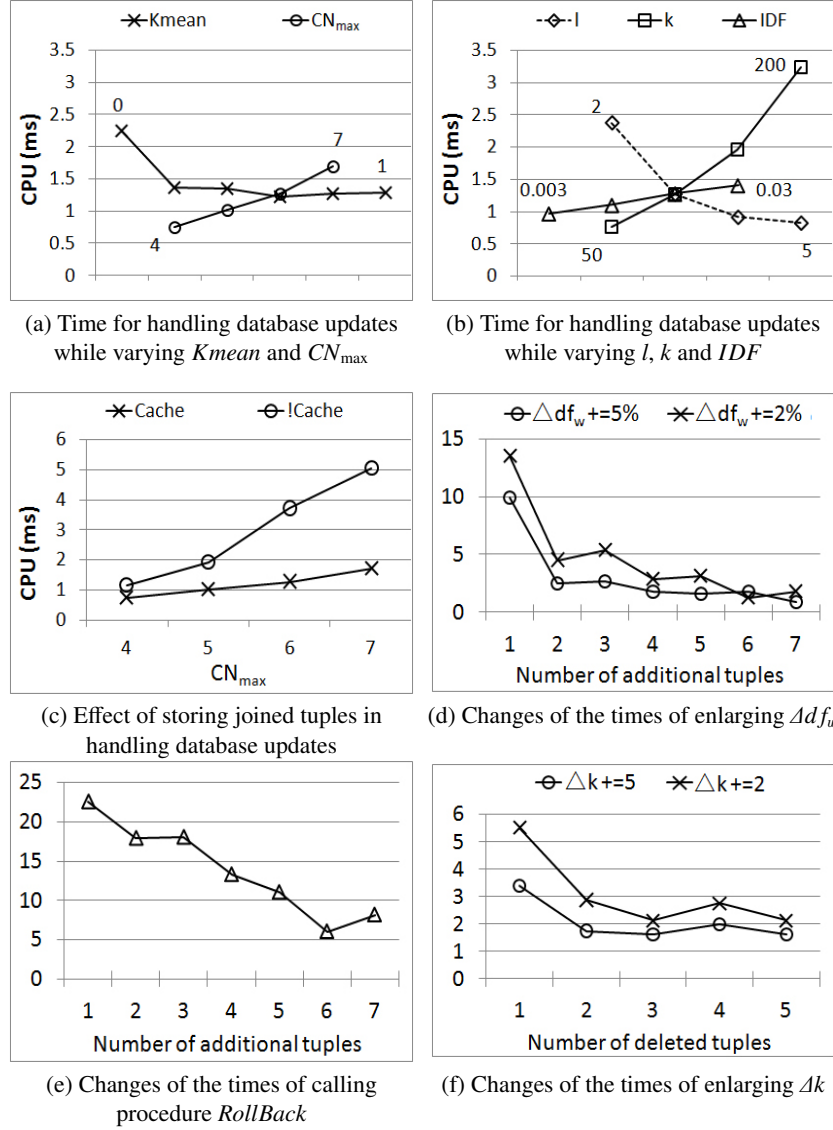
In this experiment, we want to study the efficiency of Algorithm 2 in maintaining top- k results. We use the same keyword queries as Exp-1. After calculating the initial top- k results for them, we sequentially insert additional tuples into the database by retrieving data from the DBLP XML file. At the same time, we delete randomly selected tuples from the database. Algorithm 2 is used to maintain the top- k results for the queries while the database being updated. The database update records are read from the database log file; hence the database updating rate has no directly impact on the efficiency of top- k results maintenance because the database is updated by another process.

We first add 713,084 new tuples into the database and delete 250,000 tuples from the database. The new data is roughly 90 percent of the data used in Exp-1. The composition of the additional tuples is shown in Table. 6. Fig. 11(a) and (b) show the change of the average execution times of Algorithm 2 in handling the above database updates when varying the five parameters⁴, which presents the efficiency of Algorithm 2. Note that the units for the x -axis are different for the five measures, whose minimum and maximum values are labeled in Fig. 11(a) and (b), and their other values can be found in Table. 3. We can find that the time cost of handling database updates for the default queries is smaller than 1.5ms. Comparing Fig. 11(a) and (b) with the curves of measure T in Fig. 10 (especially the curves in Fig. 10(d) and Fig. 10(e)), we can find that the time cost to handle database updates and the time cost to compute the initial top- k results have the same changing trends. This is because there are more outputted tuples in the lattice when more time is needed to compute the initial top- k results; hence more time is required to do the selections in procedures *Insert* and *Delete* and the recursive depths of them are more larger. Fig. 11(c) compares the time cost of our method in handling database updates with that of *KDynamic*, while varying CN_{\max} . The time cost of *KDynamic* is also denoted as “!Cache”. We can find that caching the joined tuples for each tuple can also improve the efficiency of handling database updates, and the larger the CN_{\max} , the higher the improvement of the efficiency is.

Table 6. Composition of the additional tuples

Papers	PaperCite	Write	Authors	Proceedings	ProcEditors	ProcEditor
156,965	20,010	411,109	111,094	3,033	3,886	6,987

⁴ Since it is hard to read in one figure, we split the data of the five parameters into two figures.

**Fig. 11.** Efficiency of top- k result maintenance

Secondly, we only insert the 713,084 additional tuples into the database while maintaining top-100 results for the default ten keyword queries. We adopt two different growing rates of Δdf_w : $\Delta df_w+ = 2\%$ and $\Delta df_w+ = 5\%$, which mean that when a $\ln\left(\frac{N}{df_w+1}\right)$ exceed its upper bound, the corresponding Δdf_w value is increased by 2% and 5%, respectively. After inserting each 100,000 additional tuples, we record the average frequency of enlarging Δdf_w and calling the procedure *RollBack* for the ten queries, whose changes are shown in Fig. 11(d) and (e), respectively, whose x -axis (with unit of 10^5) indicate the number of additional tuples. Note that we do not report the frequency of enlarging $avdl$ because it is very small in the experiment (< 2).

Fig. 11(d) shows rapid decrease after inserting the first 100,000 additional tuples. Although the frequency of enlarging Δdf_w is larger when the growing rate of Δdf_w is lower, after inserting 300,000 additional tuples, the times of enlarging Δdf_w , i.e., the times of exceeding the upper bound of $\ln\left(\frac{N}{df_w+1}\right)$, falls below 5 for both the two growing rates of Δdf_w . After inserting 300,000 additional tuples, the maximum Δdf_w value of all the relations is 15; hence it is reasonable to set 15 as the maximum value for Δdf_w . There is only one curve in Fig. 11(e) because the growing rate of Δdf_w has no great impact on the times of calling the procedure *RollBack*, which is mainly affected by the frequency of finding new results that are with $score^u > \mathcal{L}.\theta$. Note that $\mathcal{L}.\theta$ is increased after each time of calling the procedure *RollBack*. Therefore, the times of calling the procedure *RollBack* is decreasing since it is more and more harder to find new results that are with $score^u > \mathcal{L}.\theta$. In order to study the impact of reversing the pipelined evaluation on the efficiency of handling database updates, we also redo the experiment without calling the procedure *RollBack*. Then, the average time cost of handling database updates is increased by 45.4%, which confirms the necessity of reversing the pipelined evaluation.

Then, we delete 500,000 randomly selected tuples from the database after inserting the 713,084 additional tuples. Two different Δk growing rates are adopted: $\Delta k+ = 2$ and $\Delta k+ = 5$, which mean that when the number of results that are with $score^u > \mathcal{L}.\theta$ falls below k , the corresponding Δk value is increased by 2 and 5, respectively. We record the average times of enlarging Δk of the ten queries after deleting each 100,000 tuples, whose changes are shown in Fig. 11(f). Fig. 11(f) shows that the frequency of shortage of top- k results falls below a very small number after deleting 200,000 tuples, i.e., after Δk being enlarged to about 20. As indicated by the curve of k in Fig. 11(b), a large Δk value can highly decrease the efficiency of handling database updates. Therefore, it is reasonable to set the maximum value of Δdf_w as 20%.

6 Conclusion

In this paper, we have studied the problem of finding the top- k results in relational databases for a continual keyword query. We proposed an approach that finds the answers whose upper bounds of future relevance scores are larger than a threshold. We adopt an existing scheme of finding all the results in a relational database stream, but incorporate the ranking mechanisms in the query processing methods and make two improvements that can facilitate efficient top- k keyword search in relational databases. The proposed method can efficiently maintain top- k results of a keyword query without

re-evaluation. Therefore, it can be used to solve the problem of answering continual keyword search in databases that are updated frequently.

Acknowledgments

This research was partly supported by the National Natural Science Foundation of China (NSFC) under grant No. 60873040, 863 Program under grant No. 2009AA01Z135. Jihong Guan was also supported by the “Shu Guang” Program of Shanghai Municipal Education Commission and Shanghai Education Development Foundation.

References

1. S. Agrawal, S. Chaudhuri, and G. Das, “DBXplorer: Enabling keyword search over relational databases,” *ACM SIGMOD*, p.627, 2002.
2. V. Hristidis and Y. Papakonstantinou, “DISCOVER: Keyword search in relational databases,” *VLDB*, pp.670–681, 2002.
3. B. Aditya, G. Bhalotia, S. Chakrabarti, A. Hulgeri, C. Nakhe, and Parag, “BANKS: Browsing and keyword searching in relational databases,” *VLDB*, pp.1083–1086, 2002.
4. V. Hristidis, L. Gravano, and Y. Papakonstantinou, “Efficient IR-style keyword search over relational databases,” *VLDB*, pp.850–861, 2003.
5. V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar, “Bidirectional expansion for keyword search on graph databases,” *VLDB*, pp.505–516, 2005.
6. H. He, H. Wang, J. Yang, and P.S. Yu, “Blinks: ranked keyword searches on graphs,” *ACM SIGMOD*, New York, NY, USA, pp.305–316, ACM, 2007.
7. F. Liu, C. Yu, W. Meng, and A. Chowdhury, “Effective keyword search in relational databases,” *ACM SIGMOD*, pp.563–574, 2006.
8. G. Li, X. Zhou, J. Feng, and J. Wang, “Progressive keyword search in relational databases,” *ICDE*, pp.1183–1186, 2009.
9. Y. Luo, X. Lin, W. Wang, and X. Zhou, “SPARK: Top- k keyword query in relational databases,” *ACM SIGMOD*, pp.115–126, 2007.
10. G. Li, B.C. Ooi, J. Feng, J. Wang, and L. Zhou, “EASE: An effective 3-in-1 keyword search method for unstructured, semi-structured and structured data,” *ACM SIGMOD*, pp.903–914, 2008.
11. Y. Xu, Y. Ishikawa, and J. Guan, “Effective top- k keyword search in relational databases considering query semantics,” *APWeb/WAIM Workshops*, pp.172–184, 2009.
12. Y. Luo, SPARK: A Keyword Search System on Relational Databases, Ph.D. thesis, The University of New South Wales, 2009.
13. J.X. Yu, L. Qin, and L. Chang, “Keyword search in relational databases: A survey,” *Bulletin of the IEEE Technical Committee on Data Engineering.*, vol.33, no.10, 2010.
14. A. Markowetz, Y. Yang, and D. Papadias, “Keyword search on relational data streams,” *ACM SIGMOD*, pp.605–616, 2007.
15. L. Qin, J.X. Yu, and L. Chang, “Scalable keyword search on large data streams,” *VLDB J.*, vol.20, no.1, pp.35–57, 2011.
16. L. Qin, J.X. Yu, L. Chang, and Y. Tao, “Querying communities in relational databases,” *ICDE*, pp.724–735, 2009.
17. P. Jaehui and L. Sang-goo, “Keyword search in relational databases,” *Knowledge and Information Systems*, 2010.
18. A. Burns, “Preemptive priority-based scheduling: An appropriate engineering approach,” *Advances in real-time systems*, pp.225–248, 1995.

19. J.X. Yu, L. Qin, and L. Chang, *Keyword Search in Databases*, Synthesis Lectures on Data Management, Morgan & Claypool Publishers, 2010.
20. L. Qin, J.X. Yu, L. Chang, and Y. Tao, "Scalable keyword search on large data streams," *ICDE*, pp.1199–1202, 2009.
21. S.P. Lloyd, "Least squares quantization in pcm," *IEEE Transactions on Information Theory*, vol.28, pp.129–136, 1982.